APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: STREAMING OF ARCHIVE FILES.

APPLICANTS: DANNY HENDLER,
AVISHAI SHOSHANI,
URI RAZ,
YEHUDA VOLK,
SHMUEL MELAMED

Express Mail Mailing Label Number: EK839396615US

Date of Deposit December 29, 2000
I hereby certify under 37 CFR 1.10 that this correspondence is being deposited with the United States Postal Service as **"Express Mail Post Office to Addressee"** with sufficient postage on the date indicated above and is addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231.

Deborah Roberts

STREAMING OF ARCHIVE FILES

CROSS-REFERENCE(S) TO RELATED APPLICATIONS

This application is a continuation-in-part of U.S. Patent Application serial number 09/120,575, filed on July 22, 1998, entitled "Streaming Modules."

BACKGROUND OF THE INVENTION

In a client-server environment, a client computers can communicate with a server to remotely access information stored at the server. The transfer of information between the server and client computer may be provided using standard protocols and software applications. For example, a hypertext markup language (HTML) browser application at a client computer can communicate over the public Internet using TCP/IP and hypertext transfer protocols (HTTP) to receive web pages from a HTTP server. Web pages may include formatted text as well as multimedia elements, such as embedded graphics and sounds. The multimedia elements may be downloaded by the client and presented to a user by a browser application or a "plug in" browser component. Example browser applications include Netscape Navigator 4.0® and Microsoft Internet Explorer 4.0™.

Browser applications used at client computers can use plug-in software to receive audio and video information using a streaming data transmission protocol. A streaming protocol allows information to be presented by a client computer as it is being received. For example, full-motion video can be sent from a server to a client as a linear stream of frames. As each frame arrives at the client, it can be displayed to create a real-time full-motion video display. Audio and video streaming allows the client to present information without waiting for the entire stream to arrive at the client application. Audio and video streaming are provided by, for example, the RealAudio® and RealVideo™ applications from RealNetworks, Inc.

Browser applications may also make use of executable software applets to enhance the appearance of HTML-based web pages. Applets are software programs that are sent from the server to the client in response to a request from the client. In a typical applet use, HTML-based web pages include HTTP commands that cause a browser application to request an applet from a server and to begin execution of the applet. The applet may thereafter interact with a user to gather and process data, may communicate data across a network, and may display results on a computer output device. Applets may be constructed from a programming language which executes in a run-time environment provided by the browser application at the client computer. For example, the Java® programming language from Sun Microsystems, Inc., allows Java applets to be stored at a web server and attached to web pages for execution by a Java interpreter. Java Applets, may be formed from multiple Java Classes. Java Classes include executable Java code that can be downloaded from a server in response to a dynamically generated request to execute the class (a module execution request). If a Java Class is not available to a Java interpreter when an executing applet attempts to access functionality provided by the Class, the Java interpreter may dynamically retrieve the Class from a server. Other programming languages, such as Microsoft Visual Basic® or Microsoft Visual C++®, may also be used to create applet-like software modules, such as Microsoft ActiveX™ controls.

The Java programming environment may be configured for a variety of different platforms. Desktop and server computers, having relatively abundant processing and memory resources, may implementation a "full" Java architecture, such as the Java 2 Enterprise Edition, while resource limited devices may implement a more narrowly focused architecture, such as the Java 2 Micro Edition (J2ME) architecture. J2ME is a modular and scalable Java architecture that supports a minimal configuration of a Java virtual machine and Java Application Programming Interfaces (APIs). J2ME can be further targeted to particular applications and device types. J2ME defines device configuration and profile to address different market segments. A profile addresses demands of a certain market segment or device family and functions, primarily, to guarantee interoperability within a certain device family. The Mobile Information Device Profile (MIDP) is one such profile. The MIDP is designed for cell phones and related services. Additional information on the MIDP profile can be obtained from Sun Microsystems, Inc. A J2ME application is written "for" a particular profile, and a profile is based upon, or extends, a particular configuration. Thus, all the features of a configuration are included in a profile and may be used by applications written for that profile. The Connected, Limited Device Configuration (CLDC) is a configuration defined for the J2ME environment. The CLDC is targeted to personal, mobile, connected information devices. The CLDC includes a number of classes designed to server the needs of small-footprint (i.e., limited-resource) devices.

Java applets and class files can be stored in Java Archive ("JAR") files. A JAR file may be requested by a web browser using an APPLET tag embedded in a HTML page. To request the JAR file, the APPLET tag includes a 'archive' parameter identifying the JAR files that are needed by an applet. Once an archive file is identified, it is downloaded and separated into its components. During the execution of the applet, when a new class, image, or audio clip is requested by the applet, it is searched for first in the archives associated with the applet. If the file is not found amongst the archives that were downloaded, it is searched for on the applet's server, relative to the Java applet's Codebase. A different mechanism is defined by the CLDC to enable downloading of JAR files used in a CLDC device.

To enable dynamic downloading of 3rd party applications and content, the J2ME CLDC requires that an implementation support the distribution of Java applications via JAR files. Whenever a Java application intended for a CLDC device is publicly distributed over a network, it must be formatted in a compressed JAR file and classfiles within the JAR file must contain the stackmap attribute.

Downloadable applets can also be used to develop large and complex programs. For example, a complex financial program may be constructed from a collection of applets. In such a financial program, separate applets may be used to gather information from a user, compute payments, compute interest, and generate printed reports. As particular program functions are required by a user, the applets associated with the required functions can be retrieved from the server. However, as the size of a software application increases, delays associated with retrieving is modules over a network likewise increase and may be unacceptable to end-users. Consequently, an improvement in the transmission of software modules between computers is desirable.

SUMMARY OF THE INVENTION

In general, in one aspect, the invention features a method of streaming an archive file (such as a Java Archive file) from a server to a client device. The method includes extracting files from a Java Archive and streaming the extracted files from a server to a client device, receiving the streamed files and storing the received files for access by a Java application. The stored files may then be provided to a Java application

Implementations may include one or more of the following features. The client device may be a resource-limited device that includes a Java Platform Micro Edition execution environment. The received streamed files can be stored at the client device in a Java Archive format. This may help to conserve resources at the client device. Streamed files can include both executable files (e.g., Java class files) and non-executable files (e.g., image files, sound files, and other data). The files can be streamed in accordance with predictive critiera (i.e., predetermined criteria predicting an order of utilization of the files at the client device.) Meta-information in the Java archive (i.e., the Java Archive manifest and signature files) can also be streamed to the client device and used to validate other ones of the streamed files. The validation may, in a simple case, be a confirmation of file name and path. In a more complex case, digital signature information may be checked.

In general, extracting of the files occurs independent of a request by the Java application for the particular files. When the Java application does request the archive (or files in the archive), the Java execution environment at the client device will attempt to satisfy this request using the streamed files stored at the client device. If it is unable to do so, control data can be sent to the streaming server to obtain the required archive and/or required files. The control data may cause the streaming server to interrupt a file currently being streamed or to change the order of files being transmitted so that transmission of the requested file may begin.

The invention may be applied to other program execution environments and archive formats to enable streaming of files stored within different archive types. The invention may also be applied to the streaming of non-executable data.

In general, in another aspect, the invention includes methods and systems for streaming data modules between a first and a second computer. The modules may be streamed regardless of the existence of a "natural" order among the modules. For example, unlike streaming applications that rely on a natural linear ordering of data to determine the data stream contents, the disclosed streaming mechanism is not constrained to operate according to a linear data ordering. Instead, streamed data modules are selected using predetermined criteria that can be independent of the particular data content.

In an exemplary application, the disclosed streaming mechanism can provide user-dependent streaming of software modules. For example, a home banking application may include modules #1 through #5. A first banking application user may, based on the user's input choices at a menu screen, access the modules in the order 1-3-4-5 while a second user may access the modules in the order 2-4-1. For such a banking application, the predetermined criteria used to determine a streaming sequence may detail each user's module usage pattern. Predetermined criteria associated with the application's users may indicate a preferred streaming sequence 1-3-4-5 when the first user is accessing the banking application but may indicate the preferred sequence 2-4-1 when the second user is accessing the application. The streamed sequence may therefore conform to a historical user-dependent access pattern. Other types of predetermined criteria may also be used. The disclosed streaming mechanism may also be use to stream non-executable data such as hypertext markup language data, binary graphics, and text.

In general, in one aspect, the invention features a computer-implemented method of transmitting modules from a first computer to a second computer. At the first computer, a module set is formed by selecting a sequence of modules from a collection of available modules. Each of the selected modules are associated with an application executing at the second computer. The selected modules may be transparently streamed from the first computer to the second computer. The selection of modules is made in accordance with predetermined selection criteria and is independent of the second computer's execution environment.

Implementations of the invention may include one or more of the following features. A module may include non-executable data, such as hypertext markup language data, and/or program code. The selection criteria may be stored in a streaming control database. The streaming control database may include transition records associating weighted values with transitions between selected modules in the collection. Processing of transition record information, such as by using a path determination algorithm, may be used to determine the sequence of modules. The streaming control database may include list records each of which identifies a predetermined sequences of modules. Selection of modules may be made by selecting a list record. Selecting a sequence of modules may include sending data from the second computer to the first computer to identify each module in the sequence or to identify the status of the executing application. For example, data identifying the status may include a series of user input values.

Implementations may also include one or more of the following features. Streaming of the module set may be interrupted, a second sequence determined, and streaming of the second sequence may occur. The streaming of the module set may be interrupted by a request for a particular module that is sent from the second computer to the first computer. For example, a Java Applet may interrupt a stream of Java Classes by attempting to access a Java Class that has not already been streamed to the second computer. A sequence of modules may be streamed and stored at the second computer independent of the executing application. That is, the executing application need not initiate streaming and need not be aware of the streaming process. Streamed modules may be subsequently integrated with the application at the second computer by interconnecting logic in a streamed module with logic in the application.

Implementations may also include one or more of the following features. The application may include an interrupt statement. Execution of the interrupt statement may transfer control to an executor program. The executor program functions in the manner of a program code debugger by responding to the interrupt statement and preventing the permanent cessation (termination) of the executing application process. The executor program may thereafter integrate logic in a streamed module with the application's logic by replacing the interrupt statement (generally, as part of a block of replacement logic) with replacement logic from the streamed module. The application may thereafter continue executing, generally by executing replacement logic that has been substituted for the interrupt statement. The application may also include a stub procedure that can be replaced by logic in a streamed module. Replacement of the stub procedure may be direct, such as by removing the stub procedure code and replacing it with logic from a streamed module, or replacement may be operative, such as by creating a link to logic in a streamed module.

In general, in another aspect, the invention features a computer program residing on a computer-readable medium. The computer program includes instructions for causing a computer to access a collection of modules associated with an application, to access a database storing module selection criteria, to form a module set by selecting a sequence of modules from the collection in accordance with the module selection criteria, and to transparently stream the module set to a second computer. Implementations of program may also include instructions for causing the computer to retrieve a first module from the collection and to send the first module to the second computer.

In general, in another aspect, the invention features a computer program residing on a computer-readable medium. The program includes instructions for causing a computer to execute an application, to transparently receive a modules associated with the executing application, to store the received module independent of the executing application, and to integrate the received module with the executing application.

In general, in another aspect, the invention features a system for transferring information modules between computers. The system includes a first computer and a second computer. The first computer includes means for executing an application, means for receiving a sequence of modules associated with the application while the application is executing, and means for integrating a first module in the received sequence with the application. The second computer includes means for storing a collection of modules associated with the application, means for selecting a sequence of modules from the collection, and means for transferring the selected sequences from the first computer to the second computer.

Implementations may include one or more of the following advantages. Delays experienced when downloading an applications, a code module, or a data modules can be can be reduced. Software and data modules can be predictively delivered to a client workstation according to a particular end user's requirements. The order in which modules are streamed from a server to a client can be dynamically determined. A collection of module delivery sequences can be associated with a particular application or user and the sequences can be dynamically updated. Module delivery sequences can be determined based on individual software usage patterns or stored statistic associated with module usage. Module streaming can be interrupted and altered during the execution of an application. Implementations may include additional or alternative advantages as will become clear from the description and claims that follow.

DESCRIPTION OF THE DRAWINGS

Fig. 1 illustrates a computer network.

Fig. 2 illustrates computer software application modules.

Fig. 3 is a directed graph, according to the invention.

Fig. 4 illustrates a server and a client, according to the invention.

Figs. 5A – 5E illustrate application code components, according to the invention.

Fig. 6 illustrates the structure of Java Archive file.

DETAILED DESCRIPTION OF THE INVENTION

Referring to Fig. 1, a wide area network 100 is shown. In the network 100, a client computer 101 can communicate with a server computer 102 by sending data over links 103 and 104 to a data network 130. The data network 130 may include multiple nodes 131-134 that can route data between the client 101 and the server 102. The client computer 101 may transmit and receive data using the TCP/IP, HTTP, and other protocols. For example, the client 101 may use the HTTP protocol to request web pages from the server 102.

Web pages and multimedia data sent from the server 102 to the client 101 may have a natural linear sequence associated with them. The natural sequence of video data may be the linear order of video frames while the natural sequence of text may be the order in which pages of text are arranged in a document. Data having a natural linear sequence can be streamed from a server to a client to minimize download delays. In a streaming system, while earlier items in a liner sequence are being processed and/or displayed, subsequent items may be downloaded to the client computer. When processing and/or display of an item is complete, processing or display or a fully received "streamed" item may quickly begin. Since receipt of a streamed item is fully or partially complete when the item is requested, a user or client application requesting the streamed item will perceive a reduced downloading delay. For example, if the first page of a document is retrieved by a user, the second page can be downloaded while the first page is being read. If the user continues reading at the second page of the document, that page will then be available at the client, such as in a cache area on a hard disk drive, and can be read without additional downloading delay.

Software execution may not follow a predictable natural linear order. Software may include jump statements, break statements, procedure calls, and other programming constructs that cause abrupt transfers of execution among sections of executing code. The execution path that is traversed during the processing of interrelated code modules (such as code segments, code classes, applets, procedures, and code libraries), will often be non-linear, user dependent, may change with each execution of the application program, and may change depending on the state of various data items. Although a natural order may be lacking, an advantageous order may be determined in which to stream modules. The order may be determined using criteria that is independent of the computer's internal architecture or internal operating system (execution environment) considerations.

Referring to Fig. 2, a software application 200 may include multiple modules "A" through "H." Modules "A" through "H" may be Java Classes, C++ procedure libraries, or other code modules that can be stored at a server. Some of the modules "A" through "H" may also be stored at the client computer, such as in a hard disk drive cache or as part of a software library stored at the client computer. When a client computer begins execution of the application 200, a first module, such as module "A," may be downloaded from the server and its execution at the client 410 may begin. As module "A" is being processed, the programming statements contained therein may branch to, for example, module "E." If Module "E" is not already resident at the client, the execution of module "A" can be suspended, module "E" can be retrieved from the server, and then the execution of module "E" code may begin. In such a scenario, a user will experience a module download delay associated with retrieving module "E" from the server.

To minimize module download delays experienced by a user, module "E" may be transparently streamed from a server to the client computer. Transparent streaming allows future module use to be predicted and modules to be downloaded while other interrelated modules "A" are executing. Referring to Fig. 4, an exemplary software architecture 400 providing transparent streaming is shown. The software architecture 400 includes a server 401 having a database 403 of stored software modules. The server 401 can transparently transmit a stream of software modules 405 over a communications link to a client computer 410. The communication link may be an analog modem connection, a digital subscriber line connection, a local area network connection, or any other type of data connection between the server 401 and client 410. As particular software modules are being executed at the client 410, additional modules are sent from the server 401 to the client 410. In a dynamic streaming implementation, the order in which modules are streamed between the server and client may be altered based on the particular client computer 410 being served, based on the user of the client computer, and based on other dynamically determined factors.

Referring to Fig. 3, the execution order of application modules "A" through "H" may resemble a directed graph 300 rather than a linear sequence of modules. For example, as illustrated by the graph 300, after module "A" is executed, execution can continue at module "B," "D," or "E." After module "B" is executed, execution can continue at module "C" or "G." The execution path may subsequently flow to additional modules and may return to earlier executed modules.

The server 401 can use streaming control information 402 to determine the order in which to stream modules from the server 401 to the client 410. The streaming control information 402 can include, for example, a predicted execution flow between software modules such as that represented by the directed graph 300. As downloaded modules are executed by the client 410, the client may send control data 415 to the server 401 to dynamically update and alter the order in which modules are streamed from the server 401 to the client 410. Control data 415 may be used to request particular modules from the server 401, to send data regarding the current execution state of the application program, to detail the current inventory of modules residing in the client's local storage 411, and to report user input selections, program execution statistics, and other data derived regarding the client computer 410 and its executing software.

Ins A1

~~The sequence of modules sent in the stream 405 from the server 401 to the client 410 can be determined using a streaming control file 402. The streaming control file 402 includes data used by the server to predict modules that will be needed at the client 410. In a graph-based implementation, the control file 402 may represent modules as nodes of a directed graph. The control file 402 may also represent possible execution transitions between the modules as vertices ("edges") interconnecting the nodes. Referring to Table 1, in a weighted graph implementation, the streaming control file 402 may include a list of vertices represent possible transitions between modules. For example, Table 1 list vertices representing all possible transitions between the modules "A" through "H" of graph 300 (Fig. 3). Each vertex in Table 1 includes a weight value indicating the relative likelihood that the particular transitions between modules will occur. In the example of Table 1, higher weight values indicate less likely transitions. The server 401 may apply a shortest-path graph traversal~~

algorithm (also known as a "least cost" algorithm) to determine a desirable module streaming sequence based on the currently executing module. Example shortest-path algorithms may be found in *Telecommunications Networks: Protocols, Modeling and Analysis*, Mischa Schwartz, Addison Wesley, 1987, § 6.

Table 1: Graph Edge Table

Edge	Weight
(A,B)	1
(A,D)	7
(A,E)	3
(B,C)	1
(B,G)	3
(C,E)	2
(C,G)	6
(D,F)	2
(E,H)	2
(F,H)	1
(G,E)	3
(G,H)	4

For example, Table 2 represents the minimum path weight between module "A" and the remaining modules of Table 1.

Table 2: Shortest Paths from Application Module "A"

From	To	Shortest Path Weight	Path
A	B	1	A-B
	C	2	A-B-C
	D	7	A-D
	E	3	A-E
	F	9	A-D-F
	G	4	A-B-G

	H	5	A-E-H

Based on the weight values shown in Table 2, the server 401 may determine that, during the execution of module "A", the module streaming sequence "B," "C," "E," "G," "H," "D," "F" is advantageous. If a particular module in a determined sequence is already present at the client 402, as may have been reported by control data 415, the server 401 may eliminate that module from the stream of modules 405. If, during the transmission of the sequence "B," "C," "E," "G," "H," "D," "F," execution of module "A" completes and execution of another module begins, the server may interrupt the delivery of the sequence "B," "C," "E," "G," "H," "D," "F," calculate a new sequence based on the now executing module, and resume streaming based on the newly calculated streaming sequence. For example, if execution transitions to module "B" from module "A," control data 415 may be sent from the client 410 to the server 401 indicating that module "B" is the currently executing module. If module "B" is not already available at the client 410, the server 401 will complete delivery of module "B" to the client and determine a new module streaming sequence. By applying a shortest-path routing algorithm to the edges of Table 1 based on module "B" as the starting point, the minimum path weights between module "B" and other modules of the graph 300 (Fig. 3) can be determined, as shown in Table 3.

Table 3: Shortest Paths from Module "B"

From	**To**	**Shortest Path Weight**	**Path**
B	C	1	B-C
	E	5	B-C-E
	G	3	B-G
	H	7	B-C-E-H

Based on the shortest path weights shown in Table 3, the server 401 may determine that module streaming sequence "C," "G," "E," and "H" is advantageous.

Other algorithms may also be used to determine a module streaming sequence. For example, a weighted graph 300 may be used wherein heavier weighted edges indicate a preferred path among modules represented in the graph. In Table 4, higher assigned weight values indicate preferred transitions between modules. For example, edges (A,B), (A,D), and

(A,E) are three possible transitions from module A. Since edge (A,B) has a higher weight value then edges (A,D) and (A,E) it is favored and therefore, given module "A" as a starting point, streaming of module "B" before modules "D" or "E" may be preferred. Edge weight values can be, for example, a historical count of the number of times that a particular module was requested by a client, the relative transmission time of the code module, or a value empirically determined by a system administrator and stored in a table 402 at the server 401. Other edge weight calculation methods may also be used.

Table 4: Preferred Path Table

Edge	Weight
(A,B)	100
(A,D)	15
(A,E)	35
(B,C)	100
(B,G)	35
(C,E)	50
(C,G)	20
(D,F)	50
(E,H)	50
(F,H)	100
(G,E)	35
(G,H)	25

In an preferred-path (heavy weighted edge first) implementation, edges in the graph 300 having higher weight values are favored. The following exemplary algorithm may be used to determine a module streaming sequence in a preferred-path implementation:

1: Create two empty ordered sets:

i) A candidate set storing pairs (S,W) wherein "S" is a node identifier and "W" is a weight of an edge that may be traversed to reach node "S."

ii) A stream set to store a determined stream of code modules.

2: Let S_i be the starting node.

3: Append the node S_i to the Stream Set and remove any pair (S_i, W) from the candidate set.

```
4: For each node $S_j$ that may be reached from node $S_i$ by an edge $(S_i, S_j)$ having weight $W_j$:
    {
    If $S_j$ is not a member of the stream set then add the pair $(S_j, W_j)$ to the candidate set.
    If $S_j$ appears in more than one pair in the candidate set, remove all but the greatest-weight $(S_j, W)$ pair from the candidate set.
    }
5: If the Candidate set is not empty
    Select the greatest weight pair $(S_k, W_k)$ from the candidate set.
    Let $S_i = S_k$
    Repeat at step 3
```

For example, as shown in Table 5, starting at node "A" and applying the foregoing algorithm to the edges of Table 4 produces the stream set {A, B, C, E, H, G, D, F}:

Table 5: Calculation of Stream Set

Iteration	{Stream Set} / {Candidate Set}
1	{A} / {(B,100) (D,15) (E,35) }
2	{A, B} / {(D,15) (E,35) (C,100) (G,35)}
3	{A, B, C} / {(D,15) (E,35) (G,35)}
4	{A, B, C, E} / {(D,15) (G,35) (H,50)}
5	{A, B, C, E, H} / {(D,15) (G,35) }
6	{A, B, C, E, H, G} / {(D,15)}
7	{A, B, C, E, H, G, D} / {(F,50)}
8	{A, B, C, E, H, G, D, F} / {}

Implementations may select alternative algorithms to calculate stream sets.

Application streaming may also be used to stream subsections of an application or module. For example, subsections of compiled applications, such as applications written in C, C++, Fortran, Pascal, or Assembly language may be streamed from a server 401 to a client 410. Referring to Fig. 5A, an application 500 may include multiple code modules such as a main code module 501 and code libraries 510 and 515. The main module 501 contains program code that is executed when the application is started. The code libraries 510 and 515 may contain header data 511 and 516 as well as executable procedures 512-514 and 517-519 that are directly or indirectly called from the main module 501 and other library procedures.

In a Microsoft Windows 95 / Microsoft Visual C++ implementation, the main code module 501 may contain a compiled C++ "main" procedure and the library modules 510 and 515 may be dynamic link libraries having compiled C++ object code procedures. Header data 511 and 516 may include symbolic names used by operating system link procedures to dynamically link libraries 510 and 515 with the main module 501. Header data may also indicate the location of each procedure within the library. In a jump table implementation, a calling procedure may access library procedures 512-514, 517-519 by jumping to a predetermined location in the header 511 or 516 and from there, accessing additional code and/or data resulting in a subsequent jump to the start of the procedure.

Data and procedures within an application's code modules and libraries may be many hundreds or thousands of bytes long. Prior to executing an application, a client may need to retrieve a lengthy set of modules and libraries. By reducing the size of the module and library set, the initial delay experienced prior to application execution can be reduced. In a streaming implementation of application 500, code within subsections of the application's code modules can be removed and replaced by shortened streaming "stub" procedures. The replacement of application code with streaming stub procedures may reduce module size and associated transmission delay. For example, referring to Figs. 5A and 5B, the code library 510 may include a header 511 that is 4 kilobytes (Kbytes) in length and procedures 512-514 that are, respectively, 32 Kbytes, 16 Kbytes, and 8 Kbytes. Referring to Figs. 5B and 5C, to reduce the size of the library 510, procedures code 512-514 may be removed from the library 510 and stored in a streaming code module database 403 at the server 401 (Fig. 4). The removed procedure code 512-514 may be replaced by "stub" procedures 522-524 resulting in reduced-size code library 530 that can be linked with application modules 501 and 515 in place of library 510. Header data 511 of library 530 can include updated jump or link information allowing stub procedures 522-524 to act as link-time substitutes for procedures 512-514.

A server 401 may provide a streaming-enabled version of application 500 to a client 410 by sending main module 501, library module 515, "streamed" library 530, and, in some implementations, a streaming support file 535 to the client 410 in response to a request for the application 500. The streaming support file 535 may include procedures accessed by the stubs 522-524 to facilitate code streaming between the server 401 and client 410. At the client 410, modules 501, 515, 530 and 535 can be linked and execution of the resulting application can begin. As the main module 501 and various called procedures are executed at the client 410, code modules stored in the database 403 can be streamed from the server 401 to the client 410. Data may be included in the stream 403 to identify stub procedures 522-524 associated with the streamed code modules. As the streamed modules are received at the client, they are integrated with the executing application.

In an appended module implementation, streamed code modules are integrated with the executing application by appending received modules to their corresponding library or code file. For example, referring to Figs. 5C and 5D, as modules 512-514 are streamed from the server to the client, they are appended to the library file 530 thereby forming an augmented library file 540. As the modules 512-514 are streamed from the server 401 and appended to the file 530, header data 511 or stub data 522-524 is updated so that the now-appended modules are accessible from a calling procedure. For example, referring to Fig. 5D, an additional "jump" may be added between each stub procedure 522-524 and its associated appended module 512-514. Alternatively, header data 511 may be updated so that procedures 512-514 are accessible in place of stubs 522-524. In a stub-replacement implementation, stubs 522-524 are replaced by procedure modules 512-514 as the modules are received from the server 401. Stub replacement may require altering or rearranging the location of the remaining stubs or procedures within a code module or library as replacement code is received. Implementations may employ still other methods of integrating streamed code with executing applications and modules.

In some scenarios, removed code, such as procedure code 512-514 which, in the example given, was replaced by stubs 522-524, may be required (called by another procedure) before it is streamed from the server 401 and integrated with the module 530. In such a case, stub code 522-524 may access streaming functions in the streaming support library 535 to obtain the required procedure. To do so, the streaming support library 535 may send control data 415 to the server 401 to request the needed procedure. In response, the server 401 can halt the current module stream 405 and send the requested module. Upon receipt of the requested module, procedures in the streaming support library 535 may be used to integrate the received module with the application and to continue with the execution of the requested module. The server may thereafter determine a new module stream based on the requested module or other control data 415 that was received from the client.

Code modules may be reduced in size without the use of stub procedures. For example, referring again to Figs. 4, 5A, 5B, and 5E, in a interrupt driven implementation, procedure code 512-514 may be removed from a code library 510 and stored in a database 403. Header information 511 as well as data indicating the size and location of removed procedure code 512-514 may then be transmitted to a client 410. The client 410 may construct a new library 550 by appending a series of interrupt statements in place of the removed procedure code 512-514. When the application 500 is executed, the code library 550 is substituted for the library 510 and execution of the program 500 may begin. As the program 500 executes, the removed procedure code 512-514 can be streamed to the client 410 and stored in a local database 411. If the application 500 attempts to execute procedure code 512-514 it may instead execute one of the interrupt statement that have replaced procedure code 512-514. The execution of the interrupt statement halts the execution of the program 500 and transfers control to a streaming executor program 416.

Executor 416 implements interface technology similar to that of a conventional run-time object code debugger thereby allowing the executor 416 to intercept and process the interrupt generated by the application 500. When the interrupt is intercepted by the executor 416, data provided to the executor 416 as part of the client execution platform (operating system) interrupt handling functionality can be used to identify the module 550 in which the interrupt was executed and the address of the interrupt code within the module. The executor 416 then determines whether procedure code 512-514 associated with the interrupt location has been received as part of the module stream 405 sent to the client. If the appropriate procedure code has been received, the executor 515 replaces the identified interrupt with the its corresponding code. For example, procedures 512-514 may be segmented into 4 Kilobyte code modules that are streamed to the client 410. When an interrupt statement is executed by the application 500, the executor 416 intercepts the interrupt, determines an appropriate 4 Kilobyte code block that includes the interrupt statement, and replaces the determined code block with a received code module. If the appropriate code module has not yet been received, an explicit request may be sent from the client 410 to the server 401 to retrieve the code module prior to its insertion in the library 550. The executor 416 may thereafter cause the application 500 to resume at the address of the encountered interrupt.

Implementations may also stream entire modules or libraries. For example, main code module 501 may be received from the server 401 and begin execution at the client 410 while code libraries 510 and 515 are streamed from the server 401 to the client 410. Integration of streamed modules with executing modules may be provided by client 410 dynamic module linking facilities. For example, delay import loading provided by Microsoft Visual C++ 6.0 may be used to integrate streamed modules 510 and 515 with executing modules 501. Dynamic linking of streamed modules may be facilitated by storing the streamed modules on a local hard disk drive or other storage location accessible by client 410 link loading facilities. In an exemplary implementation, streaming is facilitated by altering client 410 operating system link facilities such that the link facility can send control data 415 to the server 401 to request a particular module if the module is has not already been streamed to the client 401.

In a protected-memory computer system, direct manipulation of executing application code and data may be restricted. In such systems, a “kernel” level processes or procedure may be required to support integration of streamed modules with executing application. In such a case, streaming support 535 may be pre-provisioned by installing support procedures at the client 410 prior to the client’s request for the application 500.

Archive file such as CAB, tar, BINHEX, and ZIP files, including ZIP formatted Java Archive (JAR) files, can be streamed by extracting modules of data from the archive files and streaming those modules to a client terminal. In the following text, streaming of a Java Archive is detailed. The disclosed techniques may be applied to the streaming of other archive formats as well.

Java classes, as well as images, sounds, and other data files, can be aggregated together in a Java Archive files (a “JAR file”). A JAR file is a data archive using the ZIP archive format to aggregate a collection of logically separate data files.. The files in a JAR file can include Java class files, video, sound, and other data files, as well as meta-information files (the contents of which are define by Sun Microsystems Inc.’s JAR standard).

Fig. 6 shows the format of an example JAR file 600. The content of elements 611-616, 621-626, 641-647 of the file 600 is are fields defined by the ZIP specification (this specification is available from PKWARE, Inc.) These elements 611-616, 621-626, 641-647 organize and support the archiving and compression of files 601-606 stored in the archive 600. Elements 611-616 of the archive 600 are known as local file headers. Each local file header precedes a file stored in the archive 600. Local file headers specify, among other things, the name and relative path of the associated file, the last time/date the file was modified, and the compressed and uncompressed size of the file. The contents of each local file header is repeated (together with additional information) in a central directory 640 located at the end of the ZIP archive. For each local file header 611-616 in the archive, the central directory 640 includes a corresponding central directory headers 641-646. According to the ZIP format specification, each file 601-606 in the archive may also include an optional data descriptor 621-626; though these data descriptors are typically not present in JAR files.

A ZIP file's central directory 640 includes an 'end of central directory' record 647. The record 647 includes, among other things, the total number of entries in the central directory (i.e., the total number of files in the archive), the size of the central directory, the number of the disk containing the central directory, and the offset of the start of the central directory with respect to the start of an archive segment containing the central directory 640 (a ZIP archive may be split among multiple segments, e.g., multiple floppy disk of files).

The size of a ZIP archive 600 can be obtained in a number of different ways. One method is to add the compressed sizes of the files 601-606 (obtained from the compressed file size information in either the local file header or central directory header), together with the sizes of the headers 611-616, 621-626, 641-647. A second method (usable if the ZIP archive is contained within one file or disk, as is the case with JAR files), determines the size of the ZIP archive from information in the end of central directory record 647. In particular, the value in a 'size of the central directory' field of the record 647 is added to the value of the 'offset of start of central directory with respect to the starting disk number' field in the record 647.

Files 601-606 are the files being archived using the ZIP compression technique and format. Files 601-606 include files used by an executing application 603-606 as well as meta-information files 601-602 containing information about one or more of the other files 603-

606. The meta-information includes a "manifest" file 601. Conventionally, this manifest file is named "MANIFEST.MF" and is logically associated with the storage path "META-INF" (i.e., the file has a relative path/name "META-INF/MANIFEST.MF"). The manifest file 601 includes, among other things, a JAR format version number and the names of the files 603-606 that are in the JAR 600. A manifest file also may includes optional "digest" data for one or more of the files 603-606. Digest data is generated by processing a file (e.g., file 603) with a cryptography algorithm (typically, the MD5 or SHA algorithm). The digest data is stored in the manifest file and serves as a digital signature that can be used to validate the file 603 (i.e., to ensure that the file has not been modified or corrupted). In addition to file validation, the digest information, along with separately stored signature information 602, can be used to identify a file's signer. Signature files are another form of meta-information that can be stored in a JAR. Java Archives are more fully described in, e.g., the text *Java in a Nutshell*, by David Flanagan, O'Reilly, 1997. In resource limited devices, such as J2ME / CLDC / MIDP devices, simplified meta-information, excluding signature files 602, is used.

Conventionally, a JAR file 600 is transmitted as a single file from a server 401 to a client device 410. However, the streaming server 401 may stream the file 600 as a series of separate modules by extracting the ZIP central directory 640, as well as the individual files and their associated headers 631-636, and streaming each of these elements 631-636, 640 separately. Typically, the central directory information 640 will be streamed first, followed by the meta-information 631-632 (if present), and then the remaining modules 633-636. In some implementations, the modules 631-636 or a subset, e.g., modules 633-636, will be ordered by a predictive algorithm, however such ordering is not required in all implementations. Predictive ordering may be applied to a subset 633-636 of the modules and a fixed ordering to other modules 631-632, 640.

Referring back to Fig. 4, to stream a JAR file 600, the server 401 extracts the contents of the JAR file as a set of streamable modules 631-636, 640 (other divisions and combinations of the file 600 can be used; for example, each element 601-606, 611-616, 621-626, 641-647 may be a separate module). To extract the modules 631-636, 640, their size and location is determined based on the offset of the module within the file 600, the compressed size of the files 601-606, the size each local header 611-616 and the size of each data descriptor 621-626 (if present). This size and offset information can be determined from the central directory file headers 641-647. The extracted modules 631-636, 640 are then ordered for streaming to the client 410.

The receiving device 410 includes a streaming executor 416 that controls the receipt of the streamed modules 631-636, 640 and the storage of those modules in storage 411 (which may be, e.g., solid state memory such as RAM or EEPROM, or magnetic memory such as a hard disk). The stored modules may then be accessed by a Java application or applet. The component 416 can incrementally rebuild a JAR file at the receiving device 410 to store the received modules. For example, a cell phone, PDA, or other device supporting the Java Connected Limited Device Configuration (CLDC) on the Java 2 Micro Edition (J2ME) architecture, may store the received modules in an incrementally rebuilt JAR file in memory 411. Storing the received modules in a JAR file can provide for more efficient use of limited memory resources.

To reconstruct the JAR file 600, the receiving device 410 may pre-allocate space for the file 600. The size of the archive can be determined from the end of central directory record 647. To enable space allocation for the file 600, streaming of the modules 631-636, 640 may begin with streaming of the central directory module 640 (or, at least, of the end of central directory record 647). Upon receipt of the record 647, the client device can process the size and offset information in the record 647 to determine the total size of the JAR archive 600. The client may then allocate space for the JAR archive 600. Alternatively, control data 415 can be sent to the client 410 identifying the JAR archive to be streamed and the size of the archive.

After allocating space for the JAR archive, the client places the module 640 in the allocated space at the proper offset within that space (as indicated by the offset information in the record 647). The remaining modules 631-636 may then be streamed from the server 401 to the client 410. In some implementations, streaming of the modules 631-636 may be in accordance with a predictive algorithm. As the modules 631-636 are received at the client 410 from the server 401, they can be consecutively placed in the space allocated for the JAR file. In some cases, this may result in a different ordering of the modules 631-636 in the reconstructed JAR file than the ordering of the modules in the original JAR file. For example, if module 636 is streamed before modules 631-635, then module 636 will be located at the beginning of the reconstructed JAR archive and the remaining modules 631-

635 thereafter. If such a consecutive placement of received modules in the reconstructed JAR file is used, resulting in a different order of modules 631-636 in the reconstructed file than in the original file, then the 'relative offset of local header data' contained in the module's corresponding central directory header 641-646 will need to be adjusted to reflect the new placement of each module. Alternatively, the 'relative offset of local header data' can be used to position modules 631-636 in their original position regardless of the streaming order of the modules 631-636.

Java programs executing at the client may begin using a module 633-636 following the receipt of that module; that is, the client does not need to receive all of the modules 631-636, 640 before received modules can be used. Instead, data in the fully or partially reconstructed JAR file may be extracted and de-compressed as needed using decompression and file management functionality present at the receiving device.

The client device's file extraction code may be modified to generate an exception if a module requested by an executing application is missing from the archive. Thus, if the Java application request file 616 from the archive 600, and that file has not yet been received and placed in the client's reconstructed archive, the extraction software can generate an exception. In response to this exception (which may be implemented as a procedure call to executor 416 routines), the executor 416 sends control data 415 to the server 401 identifying the requested module or file (i.e., module 636 or file 606). The server 401 may then change the streaming order of modules so that the requested module is streamed next. In some cases, a currently streaming module may be interrupted to allow immediate transmission of the requested module. This may result in discarding of the partially received module at the client. Alternatively, the requested module may be sent after any partially streamed modules. Interruption of a partially streamed module may be determined based on a weighing algorithm (e.g., finish if greater than 50% already transmitted, discard otherwise).

Java implementations may require the presence of the manifest file 601 at the client before any of the files 603-606 can be used. In such implementations, it is desirable to stream the manifest file 601 before the files 603-606. Similarly, signature files 602, if used, should be streamed prior to, or immediately following, the respective signed file 603. Streaming of the manifest file 601 and other meta information 602 in this manner permits checking and validation of the manifest information as well as the processing of digital signature information as the files 603-606 are received.

In some implementations, redundant information in an archive file may be omitted prior to streaming. For example, the local file header data 611-616 is repeated in the central directory headers 641-646; consequently, the local file header data 611-616 may be discarded from the streamed modules (resulting in a smaller module), and can be reconstructed at the client using the data in the central directory headers 641-646.

For some data types, improved compression may be obtained using type-specific compression algorithms (rather than standard ZIP compression algorithms). Prior to streaming a module, the module may be decompressed and re-compressed into a new compression format. In such cases, additional software functionality (i.e., additional decompression algorithms and ZIP compression) may be implemented at the receiving device for the reconstruction of the JAR file.

If the receiving device 410 is not resource constrained, then the streamed files also may be stored at the receiving device in a de-archived format. For example, each received file may be stored as a separate files on a hard disk 411 (or other type of memory 411). Path information in the local or central directory file headers 611-616, 641-646, or sent in control data 415, can be used to define a hierarchical storage of the streamed files at the receiving device.

In some cases, an extracted file may be further subdivided before streaming. For example, file 601 may be a class file containing several Java classes. Individual classes may be extracted from the file 601 and streamed as separate modules. This extraction of classes from within a file 601 is an implementation of the technique discussed with respect to Figs. 5A-5E.

Other methods of determining stream sets may be used. In a list-based implementation, the streaming control file may include predetermined list of module streaming sequences. For example, the streaming control file 402 may include a module streaming sequence list associated with a first user and a second module streaming sequence list associated with a second user. Control data 415 sent from the client 410 to the server 401 may identify the current user at the client 410. Once the user has been identified to the server, the server may stream software modules in accordance with the user's associated streaming sequence list. User-based streaming data may be advantageous where a user's past behavior can be used to anticipate the order of modules to be accessed by that user.

In graph-based streaming control file implementations, the weight of edges connecting nodes may be determined statically or dynamically and may be determined based on a collection of historical usage data. For example, in a programmer-controlled implementation, a software programmer estimate the likelihood that particular transitions between nodes will occur based on the programmer's knowledge of the software code and the expected application usage patterns. Alternatively, application profiling programs may be used to gather run-time execution data recording transitions between various applets, Classes or code modules and thereby determine the likelihood that particular transitions will occur. In a client-feedback implementation, control data 415 sent from the client 410 to the server 401 during module execution is used to build a statistical database of module usage and, based on that database, determine the module streaming order.

In a client-controlled streaming implementation, streaming control data 402 may be located at the client 410 and control data 415 sent from the client 410 to the server 401 may be used to sequentially request a stream of modules from the server. For example, while the client computer 410 is executing a first module, a background process may send control data 415 to a server to request additional modules that can be buffered on a hard disk 411 at the client computer 410. A client-controlled streaming implementation may used existing HTTP servers and HTTP protocols to send request from the client 410 to the server 401 and send software modules from the server 401 to the client 410. Furthermore, although streaming of software modules has been emphasized in the foregoing description, non-executable data, such as hypertext markup language, binary graphic files, and text, may be streamed as a collection of modules.

Implementations may include a "handshaking" procedure whereby, at the start of application execution, control data 415 is sent between the server 401 and the client 410. The handshaking data may include an inventory of application modules residing at the client and at the server. Such handshaking data allows both the client 410 and server 401 to determine their respective software module inventory and to optimize the stream of software modules based on that inventory information.

In a history-dependent implementation, a server or client can store data about a series of transitions between modules and calculate a new module stream based on a history of transitions. For example, referring to Fig. 3, if the module "G" was reached by the path A-B-G, then a server or client may determine that module "E" followed by "H" is to be streamed. On the other hand, if the module "G" was reached by the path A-B-C-G then the streaming sequence may include only the module "H."

The invention may be implemented in computer hardware, firmware, software, digital electronic circuitry or in combinations of them. Apparatus of the invention may be implemented in a computer program product tangibly embodied in a machine-readable storage device for execution by a programmable processor; and method steps of the invention may be performed by a programmable processor executing a program of instructions to perform functions of the invention by operating on input data and generating output.

The invention may advantageously be implemented in one or more computer programs that are executable on a programmable system including at least one programmable processor coupled to receive data and instructions from, and to transmit data and instructions to, a data storage system, at least one input device, and at least one output device. Each computer program may be implemented in a high-level procedural or object-oriented programming language, or in assembly or machine language if desired; and in any case, the language may be a compiled or interpreted language. Suitable processors include, by way of example, both general and special purpose microprocessors. Generally, a processor will receive instructions and data from a read-only memory and/or a random access memory.

Storage devices suitable for tangibly embodying computer program instructions and data include all forms of non-volatile memory, including by way of example semiconductor memory devices, such as EPROM, EEPROM, and flash memory devices; magnetic disks such as internal hard disks and removable disks; magneto-optical disks; and CD-ROM disks. Any of the foregoing may be supplemented by, or incorporated in, specially-designed ASICs (application-specific integrated circuits).